Requirements Reconnoitering at the Juncture of Domain and Instance

Martin S. Feather

USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292
Email: feather@isi.edu

Abstract

The ability to rapidly reconnoiter requirements, that is, construct, critique, contrast and complete a system's requirements would be highly beneficial. We (and others) have previously argued that this necessitates dealing with the inherent nature of requirements (their incompleteness, inconsistency, ambiguity, etc.). Here we show how such explorations can be sustained by linking a network of domain requirements with instantiations of those requirements for a particular instance of that domain.

1: Introduction

Several of us have previously argued that the distinguishing feature of requirements engineering is that it deals predominantly with the incorrect, and hence that it is primarily a process of getting right from wrong [4, 5]. The nature of requirements involves a mix of incompleteness, inconsistency, ambiguity, redundancy, non-uniformity and heterogeneity - properties we have termed 'requirements freedoms'. Thus any method or tool intended to support requirements engineering must not only be tolerant of these characteristics, but must play a positive role in assisting the analyst toward resolving them in the transition toward a specification and design for the task in question. The Requirements Apprentice project, conceived of to address these issues, is an early example of this [11].

A theme repeatedly in much of the work on requirements is the focus on *domains*. Requirements are established for a given domain of systems, and organized into a structure to capture and distinguish the various forms of a domain's requirements - those common to all systems in that domain, those that distinguish among alternative variations of such systems, policies for achieving the requirements, etc. Examples include the cliche library of the Requirements Apprentice [10], the domain goal relations graph of Oz [12], the goal structures that are instances of the KAOS meta-model [2].

In this paper I show how a simple network representation of a domain's requirements can be used to support the rapid reconnoitering of a prospective (or existing) system's requirements. This representation is quite robust with respect to the requirement freedoms identified above. Queries to check for inconsistency and incompleteness have been studied in the broader realm of project management, e.g., [9, 13]. This effort can be seen as tailoring a simplified version to dealing only with requirements.

The paper is organized as follows: Section 2 presents a simple network representation for requirements, implemented within an entity-relationship database. This is illustrated on the domain of resource management, and a particular instance of that domain. Section 3 shows how queries of such a representation can support requirements analysis activities dealing with incompleteness, inconsistency, and comparison of alternatives. Section 4 draws some observations from this exercise.

2: Representing requirements as a network

In this section I describe a simple network representation of requirements. The essence of this representation is an entity-relationship model used to express requirements. *Entities* comprise:

- Goal - an objective of the overall domain
- Policy - a typical means of achieving a goal or goals

and *Relationships* comprise:

- Supports - a policy may support a goal
- Impedes - a policy may impede a goal
- Augments - a policy may augment another policy
- SubGoalOf - a goal may be a subgoal of another goal

This model is used to represent domain requirements, that is, the goals, policies and their inter-relationships common to (or alternatives among) the class of systems characterized by that domain. The following additions support representation of systems that are instances of that domain:

0-8186-3120-1/92 $3.00

Entities:

- Instance - a system to be viewed as an instance of the domain

Relationships:

- PolicyInstance - a policy of the system
- ImpossiblePolicyInstance - a policy deemed impossible for this particular system

Note that there is a simple *meta-model* [2] implicit within this representation, e.g., Supports can hold only between a policy and a goal; PolicyInstance relates a policy to an instance. This meta-model is simple in comparison to that of the cited KAOS system.

Implementation: The above model is implemented in AP5, our in-house relational extension of Common Lisp [7]. The classes of entities (Goal, Policy and Instance) are represented as AP5 types, and the classes of relationships (Supports, etc.) as AP5 relations. AP5's type discipline ensures adherence to the meta-model, e.g., Supports can only be asserted to hold between a policy object and a goal object. All the queries shown in this paper have been executed with this implementation.

2.1: Example

The domain that serves as the example within this paper is that of resource-management (where users seek access to a bounded resource). I use this domain because it is readily understood without elaborate explanation of domain details, because it is sufficiently rich to be interesting, and because it can serve as a common point for comparison and contrast (this domain and instances of it have been used as illustration for other requirements research, for example, the 'lending library' problem from the IWSSD problem set [8], video rental, theatre reservation, allocation of applications for university courses...).

Using the simple model of requirements outlined above, my representation of (some aspects of) the resource-management domain is as follows:

Goals: each goal of the resource-management domain is represented as a text string, briefly expressing the goal (only we, the human reader, understand the natural language contents; my system deals only with the structure established between these strings treated as atomic objects):

"effective use of resource"; "satisfy user requests"; "do not burden user"; "discourage misuse" ; "equal access"; "load balance"; "avoid no-shows".

Policies: like goals, these are represented as text strings which the system treats as atomic objects. Policies differ from subgoals by being possible means to achieve the goals to which they are linked, whereas subgoals are used to represent a finer-grained decomposition of their parent goals. Some of the policies for resource management are:

"reserve in advance"; "replenish resource"; "allow cancellation of reservation"; "charge for reservation"; "partially refund reservation fee"; "penalize for no-show"; "waitlist"; "system can cancel reservation"

Relationships: these are used to capture the network structure among the goals and policies of this domain. The following relationships represent structure within the resource-management domain:

- SubGoalOf "avoid no-shows" "effective use of resource"
- SubGoalOf "load balance" "effective use of resource"
- Supports "waitlist" "effective use of resource"
- Supports "reserve in advance" "effective use of resource"
- Supports "charge for reservation" "discourage misuse"
- Supports "replenish resource" "effective use of resource"
- Impedes "reserve in advance" "do not burden user"
- Impedes "allow cancellation of reservation" "do not burden user"
- Impedes "system can cancel reservation" "do not burden user"
- Augments "allow cancellation of reservation" "reserve in advance"
- Augments "penalize for no-show" "reserve in advance"

Implementation: I represent the above goals and policies as objects of the corresponding AP5 types, and the above relationships as instances of the corresponding AP5 relations between the appropriate objects.

As an instance of the resource-management domain I consider the wilderness permit system that used by the U.S. Forest Service in certain areas of California to control access to wilderness areas. Basically, entry into wilderness is treated as a bounded resource - hikers who wish to enter the wilderness at a trailhead and camp overnight are required to acquire a wilderness permit prior to entry. Each trailhead is limited as to the number of such hikers allowed to enter each day, thus setting the bound on the resource. The system by which the Forest Service manages this resource is the aspect I represent as an instance of the resource-management domain.

The following policies characterizing the "Wilderness Permit" system:

"charge for reservation"; "allow cancellation of reservation" and "reserve in advance".

The policy "replenish resource" is impossible for this system.

Note that my model of system instances is very crude - for example, all domain goals are assumed to hold of the system instance, whereas in practice it is likely that only some subset of them will be desired; the model could be easily augmented to record which of the domain goals are inherited by the instance, just as domain policies can be marked as impossible for the system instance

Implementation: I use an AP5 object of type instance to represent the Wilderness Permit system, and instances of

the AP5 relations corresponding to PolicyInstance and ImpossiblePolicyInstance.

3: Reconnoitering the requirements

We can use the network representation to explore the state of the wilderness permit system with respect to the resource management domain. Explorations take the form of *queries* issued against the network. For example, the query *"which of the domain goals are supported by the wilderness permit system's policies?"* can be coded as the following query to the network:

```
(listof (g) s.t.
  (and (goal g)
       (E (p) (and (PolicyInstance p "Wilderness Permits")
                   (Supports p g)))))
```

which returns the list

("discourage misuse" "effective use of resources")

Implementation: The above is an instance of an AP5 query. Briefly, queries are constructed out of expressions in first-order logic. Thus (Supports p g) is a truth-valued expression, true if and only if the objects bound to p and g are related by the Supports relation. The usual logical connectives can be applied to build compound expressions, thus (and (Policy...) (Supports...)) builds the conjunct of its clauses. (E (p) ...) denotes existential quantification ["E" for existential] - in this case, true if there exists an object p satisfying the truth-valued expression that follows (and (Policy...) (Supports ...)). The outermost (listof (g) s.t. ...) computes the list of objects satisfying the truth-valued expression that follows (and (goal g) ...).

A variety of such queries can be issued, e.g.,

- Supported goals (query shown above)
- Supported goals *and* the corresponding policies that support them (a simple extension of the query shown above)
- *Un*supported goals - introduce a negation before the existential in the above:
  ... (not (E (p) (and (PolicyInstance ...

This network representation and its queries can be used to support the activities that deal with the variety of 'requirements freedoms' that we, and others, have identified as pervasive in requirements engineering. These issues are explored in the subsections that follow.

3.1: Inconsistency

Inconsistent requirements are common; indeed, much of the activity of requirements engineering seems to be finding an appropriate balance between idealized but incompatible requirements. Our network is tolerant of inconsistency, for example, the goals "effective use of resources" and "do not burden the user" are generally inconsistent, as reflected by the inclusion of domain policies that support the former while impeding the latter. Queries can be used to identify where these occur:

(listof (g h) s.t. (E (p) (and (Supports p g) (Impedes p h))))

returns the singleton list of one such pair:

(("effective use of resource" "do not burden user"))

We can. of course, query whether our particular system instance contains such inconsistency:

```
(listof (g h) s.t.
   (E (p) (and (PolicyInstance p "Wilderness Permits")
               (Supports p g) (Impedes p h))))
```

which returns the same singleton list.

3.2: Incompleteness

Incompleteness, like inconsistency, can be present at either or both of the domain and instance levels of our network representation. My models of the resource-management domain and of the wilderness permit system are far from complete; nevertheless, my belief is that even incomplete representations such as these can prove useful.

The knowledge that is encoded within the network can be explored from simple completeness points of view, for example, we can ask for *domain goals for which there are no policies in support of them*, in order to explore the state of completeness of our domain model. E.g.,

```
(listof (g) s.t.
   (and (Goal g)
        (not (E (p) (and (Policy p) (Supports p g))))))
```

returns the list:

("equal access" "load balance" "avoid no-shows" "do not burden user")

Likewise, we can ask for the domain goals that our system instance does not support via any policies, domain policies that we have not deemed impossible but have not included, etc.

3.3: Alternatives / comparisons

Different versions of the same system, and different systems, can be represented simultaneously as instances of the same domain model, and compared. For example, it was only in recent years, and only in certain areas, that the Forest Service introduced the policy of charging a fee for reservations. The 'other' version of this system (without such fees) can be quickly created within this model by creating another Instance object, "Free Wilderness Permits" say, and assigning it all the same PolicyInstance and ImpossiblePolicyInstance relations as "Wilderness Permits" except for the policy "charge for reservation". Once in place, the following query finds those goals that are supported by a policy of "Wilderness Permits" but not by any policy of "Free Wilderness Permits" (the results of

such a query might be more interesting in the case of two systems instances that were further apart):

```
(listof (g) s.t.
  (and (E (p) (and (PolicyInstance p "Wilderness Permits")
                   (Supports p g)))
               (not (E (p) (and (PolicyInstance p "Free
                                 Wilderness Permits")
                                (Supports p g))))))
```

returns the singleton list: ("discourage misuse")

4: Observations and conclusions

The preceding sections have shown how a very simple model of requirements for a domain, and instances of that domain, can be built as an entity-relationship network. This permits the reconnoitering of that network by issuing simple queries against the stored information. I now make some observations regarding this:

Bringing together a representation of domain requirements with a representation of instances of systems of that domain promotes the transfer of information from domain to instances, between instances, and (potentially) from instances back to the domain model.

A simple representation, backed by a query mechanism, permits the easy formulation of a wide range of queries against the accumulated requirements knowledge. This is reminiscent of (indeed, inspired by) the gIBIS work [1].

My domain model is clearly weak in many aspects. It lacks any representation of many of the requirements and policies of resource-management (e.g., there is no representation of 'confirmation' of reservations, or of whether or not rescheduling is supported). It also lacks a connection to a quantitative or behavioral model. Studies in these directions, for the purposes of critiquing software specifications, have been performed [3,6]. Finally, the relationship between the instance level and the domain models is rather trivial - it seems plausible that a more sophisticated mechanism of linkage, involving instantiation of parameters and the like, would become necessary in more complex examples. My hope is that my model could easily be extended to include more knowledge, to make use of a more sophisticated representation scheme, and to interface with more sophisticated tools and models.

I have avoided the issue of the 'process' or 'methodology' by which requirements should be acquired, validated, etc. The unrestricted nature of the model could be constrained to encode, or operate in conjunction with, such processes.

Acknowledgments

The author has benefitted particularly from the research contexts provided by ISI's Software Sciences Division, and Steve Fickas' group at the University of Eugene, Oregon, and from reviewer comments. Support from this work has been provided by Defense Advanced Research Projects Agency contract No. BAPT 63-91-K-0006; views and conclusions in this document are those of the author and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, the Forest Service, or any other person or agency connected with them.

References

[1] "gIBIS: A Tool for all Reasons," *Journal of the American Society for Information Science*, p.p. 200-213, May 1989.

[2] A. Dardenne, S. Fickas and A. van Lamsweerde, "Goal-Directed Concept Acquisition in Requirements Elicitation," in *Proc. 6th International Workshop on Software Specification and Design* (Como, Italy). IEEE Computer Society, 1991, p.p. 14-21.

[3] K. Downing and S. Fickas, "Specification Criticism Via Goal-Directed Envisionment," in *Proc. 6th International Workshop on Software Specification and Design* (Como, Italy). IEEE Computer Society, 1991, p.p. 22-30.

[4] M.S. Feather, "Requirements Engineering: Getting Right from Wrong," in *Proc. 3rd European Software Engineering Conference* (Milan, Italy). Springer-Verlag, 1991 p.p.485-488.

[5] M. S. Feather and S. Fickas, "Coping with Requirement Freedoms," in *Workshop Notes, International Workshop on the Development of Intelligent Information Systems* (Niagara-on-the-Lake, Ontario, Canada, April 1992).

[6] S. Fickas and P. Nagaranjan, "Critiquing Software Specifications," in *IEEE Expert*, November 1988, p.p. 37-47.

[7] N. Goldman and K. Narayanaswamy "Software Evolution through Iterative Prototyping," in *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, 1992.

[8] Proceedings of the 4th International Workshop on Software Specification and Design, Monterey, California, IEEE Computer Society, 1987.

[9] M.H. Penedo "Prototyping a Project Master Data Base for Software Engineering Environments," in *Sigplan Notices (Proc. 2nd ACM Sigsoft/Sigplan Symp. Practical Software Development Environments, Palo Alto, CA)*, Jan 1987, 22(1), p.p. 1-11.

[10] H.N. Reubenstein and R.C. Waters, "The Requirements Apprentice: An Initial Scenario," in *Proc. of the 5th International Workshop on Software Specification and Design*, Pittsburgh, Pennsylvania, IEEE Computer Society, 1988, p.p. 211-218.

[11] C.Rich, R.C. Waters and H.N. Reubenstein, "Toward a Requirements Apprentice", in *Proc. of the 4th International Workshop on Software Specification and Design*, Monterey, California, IEEE Computer Society, 1987, p.p. 79-86.

[12] W.N. Robinson, "Integrating Multiple Specifications Using Domain Goals" in *Proc. of the 5th International Workshop on Software Specification and Design*, Pittsburgh, Pennsylvania, 1988, IEEE Computer Society p.p. 219-226

[13] A. van Lamsweerde, B. Delcourt, E. Delor, M-C. Schayes and R. Champagne, "Generic Lifecycle Support in the ALMA Environment," in *IEEE Transactions on Software Engineering*, June 1988, 14(6), p.p. 720-739.